CORE Metadata, citation and

松 山 大 学 論 集
第27巻第４・２号抜刷
２０１５ 年 10 月 発 行

量子乱数によるモンテカルロ・シミュレーションの理論および実証研究

檀　　裕　也

量子乱数によるモンテカルロ・シミュレーションの理論および実証研究

檀　　裕　也

1　はじめに

モンテカルロ法に基づくシミュレーションでは，アルゴリズムに従ってコンピュータが計算する擬似乱数を用いて偶発的な数値計算を実行することが一般的である。そのため，これまでに著者が実行した主に社会現象のシミュレーション［1-5］は，擬似乱数の周期性と生成次元に関する制約を伴い，シミュレーションの規模や試行回数において，スケーラビリティに課題を残したままになっていた。

本研究では，これまでに明らかになったシミュレーションの結果について，真正乱数に置き換えたシミュレーションの再実行によって計算精度の向上を図ろうとするものである。コンピュータのアルゴリズムによって機械的に生成される擬似乱数の制約を超えた物理乱数として注目されている量子乱数の性質を明らかにすることが求められている。そのためには，量子乱数の統計的性質など理論的なアプローチだけでなく，実用化の段階にある量子乱数生成器を用いた実践的な研究が求められていると言える。

量子乱数の統計学的な性質を調べるとともに，従来手法である擬似乱数との本質的な相違を明らかにする。その過程において，量子力学の理論的な背景だけでなく，実際の量子乱数発生器を用いた実証研究によって，いくつかの応用事例を示したい。また，量子乱数の限界を指摘し，実際のシミュレーションに適用する際の注意を与えることが本研究の目的である。

統計学的に性質の良い量子乱数を用いたシミュレーションに関する研究の中で，理論だけでなく実証研究によって応用を目指す点に本研究の学術的な特色および独創的な点があると考えられる。また，量子力学に基づくデバイスを研究の対象とするため，量子コンピュータなど将来に向けた理論を明らかにできる可能性がある。いずれにしても，量子デバイスを実際のシミュレーションに導入することに大きな意義がある。

これまで，電気ノイズ等物理乱数を用いた実証研究は多く，その統計学的な性質が調べられているとともに，応用事例が提案されている。量子乱数を用いたものは，量子乱数発生器の開発が最近になって成功したため，研究成果が少ないという現状がある。現在，最も有効だと考えられているメルセンヌ・ツイスター擬似乱数の性能を超える真の乱数の登場が期待されている。

本研究課題は，以下の 3 つのステップに分けて，原則として順に進めていくことを計画している。その間，本研究に関連する論文および文献の調査，関連学会における調査を踏まえることによって，先行研究について網羅的に情報を収集する。

⑴ 量子乱数発生器の導入

IDQ 社が 2011 年に開発した Quantis 量子乱数発生器を導入し，乱数の統計学的性質について調べる。すなわち，メルセンヌ・ツイスター擬似乱数と比較して，真の乱数として振る舞うことを明らかにする。その際，単一光子の量子力学的な性質について調べ，本装置の出力性能を評価する。

また，同製品に付属する開発キットと連携したプログラムを開発し，その乱数をコンピュータで処理する仕組みを実装する。

⑵ モンテカルロ法による検定

量子乱数発生器によって得られた真の乱数列を用いてモンテカルロ・シミュレーションを実行するプログラムを作成する。具体的には，定積分の計算など，

理論的に結果が知られているものを乱数によって追従するものから始め，組み込み擬似乱数やメルセンヌ・ツイスター擬似乱数などのアルゴリズム乱数と収束速度を比較し，その有効性を明らかにする。その後，既存のシミュレーションプログラムについて量子乱数に置き換えた改良を施し，計算速度を比較する。

⑶　応用事例の提案

著者による先行研究の成果として得られているシミュレーションや量子多体系の性質を踏まえて，量子乱数ならではの応用事例を提案したい。現在，暗号の生成や量子暗号の生成といったネットワーク上の情報セキュリティに関する応用，PIN 番号の生成，携帯プリペイドシステム，クラウドコンピューティング等さまざまなアプリケーションが考案されている段階にあるので，実装を含めて本研究によって明らかにされた性質を最大限に発揮するような応用事例を提案したい。それと同時に，量子乱数の限界を指摘し，応用事例に関する示唆を与えたい。

2　モンテカルロ法

モンテカルロ法とは，乱数を使って数学的問題の解を求めることである。一般に，与えられた数学的問題に対して，計算をしたり，方程式を解いたりして解を求めることが多い。しかし，数学的問題の中には，そのモデルを方程式で表現できない現象や，方程式で表現できたとしても解析的な方法で解くことが不可能なものさえ存在する。そこで，コンピュータによる数値解法以外に，乱数を使って近似解を求めるモンテカルロ法が注目されている。

もともとは，J. von Neumann と S. M. Ulam が第 2 次世界大戦中に原子爆弾の開発に向けて核分裂における中性子の拡散現象をコンピュータでシミュレーションしたときに導入したと考えられている。モンテカルロの名称は，カジノで有名なモナコ公国にある都市にちなむ。[6-7]

従来の研究では，ソーシャルネットワーク上の情報の拡散現象をシミュレー

ションした研究［1-2］のほか，ストレージ上に記録されるファイルの蓄積現象をシミュレーションした研究［3］，学生の研究室配属という社会モデル［4］，その他，マーケティングや世論調査など予測に応用可能なブートストラップ法を取り入れた研究［5］がモンテカルロ法をベースにしたものである。確率変動を含む偶発事象だけでなく，確定事象に対して予測を精緻化できるといったアプローチが存在する。

3 量子乱数発生器

微細構造加工技術（ナノテクノロジー）の進歩に伴って，多くの量子デバイスが開発されるようになってきた。その中には，量子計算や量子通信を指向するものも含まれるが，今回の研究で取り上げるのは量子乱数発生器である。

すでに製品化されている IDQ-QUANTIS は，量子的な光学プロセスを利用した量子物理学に基づく真性乱数発生器である。製品には，PCI インターフェース，PCI-Express インターフェース，OEM 組み込み用に加えて，USB 接続の 4 つのタイプがラインナップされている。本研究では，PC との接続の関係から USB 接続の IDQ-QUANTIS を導入することにした。メーカーは，世界で唯一の真性乱数発生器であることを謳っている。

この量子乱数発生器の動作原理はシンプルなものである。ボード内に用意された半透明の鏡に光量子（フォトン）を一つずつ照射し，反射するのか透過するのかという排反事象を観測することによって 0 および 1 のビット値に関連づける。このようにして生成されたランダムビットストリームを出力することによって真正乱数列として取り出すことができるようになる。なお，本製品は標準で毎秒 4 M ビット（最大で 16 M ビット）のランダムビットストリームの比較的高い出力性能を持っている。

古典物理学におけるマックスウェルの電磁気の理論によると，電場と磁場の振動の結果として発生する電磁波には，進行方向に対して電磁場の振動面の向きとして偏光という現象が知られている。すなわち，垂直方向および水平方向の

２つの振動面をはじめ，振動面は任意の角度となることができる。特定の振動面を吸収する偏光板を用いると，その他の振動面の電磁波のみ透過させることができ，その結果として，単一の振動面を持つ電磁波を構成することができる。

一方，量子力学によると，フォトンは偏光について２つの状態を取ることができる。厳密にいうと，その２つの状態はヒルベルト空間における固有状態であって，任意の状態は２つの量子状態の重ね合わせとして表現することができる。いま，一方の量子状態を$|0>$，他方の量子状態を$|1>$で表すことにしよう。フォトンの状態を記述するヒルベルト空間をHとすると，$|0>\in$ H および$|1>\in$ H ならば，$|\alpha|^2+|\beta|^2=1$ を満たす任意の複素数α およびβ を用いて，２つの量子状態の重ね合わせ$\phi=\alpha|0>+\beta|1>$ はヒルベルト空間Hに属すことが分かる。量子力学によると，量子状態ϕ を観測すると，確率$|\alpha|^2$で０を観測するとともに量子状態は$|0>$に収縮し，確率$|\beta|^2$で１を観測するとともに量子状態は$|1>$に収縮する。そこで，半透明の鏡を用いて，片方の量子状態だけを透過させる使用を実装することによって，あらかじめ定められた確率で０と１のランダムビットストリームを取り出すことができるわけである。

量子乱数発生器 IDQ-QUANTIS には，上記の動作原理に基づく量子デバイスとともに，GUI インターフェースによる IDQ-EasyQuantis アプリケーションプログラムが付属し，ランダムビットストリームの読み出しおよび保存などデバイスを制御することができる。さらに，量子乱数発生器 IDQ-Quantis ライブラリには，C++，C＃，Java および VisualBasic. NET のアプリケーションプログラミングインターフェース（API）が提供されているため，独自のプログラムを開発することが可能である。その際，バイナリ，整数および浮動小数点を含めたさまざまな形でソフトウェアの実装が容易にできる仕組みとなっている。

量子乱数発生器 IDQ-Quantis のプロセスユニットには，生成されるランダムビットストリームに対して“Failure”＆“Disabling”を検出するためのモニタリング機能が備わっており，信頼性の高い真正乱数を得ることができる。また，スイス連邦政府の度量計測機関 METAS によって測定・テスト・評価・承

認されたほか，マルタの Lotteries & Gaming Authority など国家当局の承認も得ている製品であって，ゲーミングアプリケーションにも使用できると謳われている。

本製品に限らず，一般の量子乱数発生器の応用として，乱数・暗号・統計学の研究やシステム開発などが挙げられる。特に，暗号作成生成装置，量子暗号化，ネットワークセキュリティ，ギャンブル機器，宝くじ，パチンコ，オンラインゲーム，印刷物のセキュリティ，PIN 番号発生，クラウドコンピューティングシステム，携帯プリペイドシステム，統計学，数値シミュレーション，モンテカルロ法などの開発に量子乱数が貢献できる道があると考えられている。実際，ベンダーの宣伝文句として，クラウドコンピューティング，量子暗号化技術，シミュレーション，オンラインゲームなど新しい時代のネットワークインフラとセキュリティに重要な役割を果たすと表現されている。[8]

4 実 験 の 方 法

量子乱数発生器 IDQ-QUANTIS によって生成されるランダムビットストリームの性質を評価するため，簡単なモンテカルロ法に基づくシミュレーションによって定積分の計算を実行し，解析的な手法を用いて得られる解と比較する。具体的には，よく知られた定積分

$$\int_0^1 \sqrt{1-x^2}dx = \frac{\pi}{4}$$

について考察する。定積分そのものは，三角関数による置換積分法から容易に得られるため異論のないところであろう。幾何学的には，原点に中心が位置する単位円のうち，第 1 象限に限定した部分の四分円の面積を求めていることになる。

一方，この定積分はモンテカルロ法に基づくシミュレーションによってコンピュータの数値計算で近似値を求めることができる。いま，面積を求めたい四

分円と，その四分円を含む一辺が長さ 1 の正方形を考えよう。正方形の頂点の座標は，(0，0)，(1，0)，(1，1) および (0，1) である。ある点 (x，y) が四分円の内部に含まれるときの必要十分条件は

$$x^2 + y^2 < 1$$

と表すことができる。ただし，円周上の点は除く，ということになるが，数値計算において本質的な問題ではない。

ここで，点 (x，y) の座標値 x および y について，区間 [0，1) の範囲で一様な乱数によって与えることにすると，正方形の面積が 1 であることから，四分円の内部に含まれる確率は $\pi/4$ となることが分かる。そこで，モンテカルロ法に基づくシミュレーションを繰り返すことによって得られる統計情報から円周率 π の計算ができることから，π 計算の近似の精度から乱数の一様性という性能を読み取ろうと試みるわけである。

なお，次のソースコードは，四分円の内部に含まれる回数を求めるところで，関数 **myRandomNumber** は区間 [0，1) の一様乱数を返すものである。変数 **c** には，シミュレーションの試行回数 **n** のうち，四分円の内部に含まれる回数が求められる。

《四分円の内部に含まれる回数を求めるソースコードの一部》

```
c = 0;
for( i= 0; i < n; i++ ){
    x = myRandomNumber();
    y = myRandomNumber();
    if( x * x + y * y < 1 ){
        c++;
    }
}
```

4. 1　線形合同法

コンピュータによる区間［0，1）の一様乱数を疑似的に求めるアルゴリズムとして，古くから線形合同法が知られている。次のような方法で，数列 $\{a_0, a_1, a_2, \cdots\}$ を求めてみよう。初期値 a_0 は擬似乱数の種（シード）とも呼ばれているもので，どのような値を与えてもよい。ここでは，32 ビットの符号なし整数を用いることにする。線形合同法とは，漸化式

$$a_{n+1} = a \times a_n (\mathrm{mod.}\, m) \quad n = 0,\ 1,\ 2,\ \cdots$$

によって求められる無限数列を乱数とみなすアルゴリズムである。ここで，右辺のモジュロ計算は m で割った余りで計算することを意味し，32 ビットの符号なし整数の場合，$m = 2^{31} - 1 = 2147483647$ とする。$2^{32} - 1$ を用いないのは，実装上のテクニカルな理由（桁あふれの問題から等式 $2^{32} = 0$ がレジスタ上で成り立つ）による。乱数性は係数 a の与えかたに依存するところが大きいが，一般に $a = 16807$ としておくと，比較的性質の良い擬似乱数が得られると考えられている。

簡素なアルゴリズムで実装可能な線形合同法によって高速に得られる擬似乱数は，使いかたによっては大きな制約を伴う問題も存在する。また，無限数列ではあっても周期 m でシードに戻るという周期性があるため，モンテカルロ法に基づくシミュレーションでは，精度に限界があることになってしまう。そのようなメリットとデメリットを踏まえた上で，関数 **myRandomNumber** に線形合同法によるアルゴリズムで擬似乱数を生成するソースコードを実装した。

《関数 **myRandomNumber** のソースコードの抜粋》

```
double myRandomNumber()
{
    static unsigned int x = 1;
```

```
    const unsigned int a = 16807;
    const unsigned int m = 2 << 31 - 1;

    x = a * x % m;

    return (double)x / (double)m;
}
```

4.2　メルセンヌ・ツイスター（MT）法

線形合同法に基づく擬似乱数生成では，その乱数列の周期性が大きな課題として存在していた。その問題点を解消したのが，1996 年から 1997 年にかけて開発されたメルセンヌ・ツイスター（MT）法［9］である。もちろん，あるアルゴリズムに基づく擬似乱数であることから有限の周期性があるという性質は残るものの，その周期を $2^{19937}-1$ にまで大きくすることに成功した擬似乱数［10］である。

4.3　量子乱数

量子乱数は，その生成原理から，乱数列としての周期性はなく，周期は無限大であると考えられる。量子デバイス内では，単一光子（フォトン）の発生から観測までの一連の操作があるため，物理的な測定および出力の時間がかかることになる。

4.4　実験プログラム

例えば，線形合同法に基づく擬似乱数を実装した評価実験に用いたプログラムは C 言語で記述し，ソースコードは以下の通りである。その他の乱数については，関数 **myRandomNumber** のソースコードの部分だけ異なるものである。

《ソースコード》

```
#include <stdio.h>
#include <time.h>

double myRandomNumber()
{
    static unsigned int x = 1;
    const unsigned int a = 16807;
    const unsigned int m = 2 << 31 - 1;

    x = a * x % m;

    return (double)x / (double)m ;
}

int main( int argc, char *argv[] )
{
    long i, j;
    double x, y;
    long b, c, n = 1024;
    clock_t start, end;

    /* Initialization */
    printf("*** Monte Carlo pi Computation ***¥n");

    /* Monte Carlo Simulation */
    while( n > 0 ){
        b = 0;
        printf( "n = %d, ", n);
        start = clock();
        for( j = 0; j < 10; j++ ){
            c = 0;
            for( i = 0; i < n; i++ ){
                x = myRandomNumber();
```

```
            y = myRandomNumber();
            if( x * x + y * y < 1 ){
                c++;
            }
        }
        b += c;
        printf( "%d, ", c);
    }
    end = clock();
    printf( "b = %d, pi = %18.16f¥n", b, 4.0 * b / n / 10.0 );
    printf( "Average process time : %d [ms]¥n",
        ( end - start ) / 10 );
    n *= 16;
  }

  return 0;
}
```

上記のソースコードを作成したら，Microsoft Visual C++ 2010 のコマンドプロンプトからコンパイル・リンクし，バイナリ形式の実行ファイルを Microsoft Windows 7 上で実行して得られた結果は，次の通りである。なお，上記ソースコードは，特定のハードウェアやオペレーティングシステムに依存する機能は含まれていないため，どのような開発環境・実行環境であっても動作するはずである。

5　実　験　結　果

5.1　線形合同法

線形合同法によって生成された擬似乱数を用いて実装したプログラムの実行結果は，以下の通りである：

《実験結果》

```
C:¥Users¥dan>pi
*** Monte Carlo pi Computation ***
n = 1024, 811, 801, 826, 803, 816, 804, 805, 796, 796, 810, b = 8068, pi = 3.1515624999999998
Average process time : 0 [ms]
n = 16384, 12861, 12953, 12842, 12908, 12909, 12884, 12807, 12896, 12890, 12944, b = 128894, pi = 3.1468261718749999
Average process time : 2 [ms]
n = 262144, 205925, 205486, 205948, 205734, 206348, 205845, 205745, 206116, 205963, 205471, b = 2058581, pi = 3.1411453247070313
Average process time : 36 [ms]
n = 4194304, 3293778, 3293407, 3295627, 3295185, 3293738, 3294563, 3295121, 3294382, 3294223, 3293834, b = 32943858, pi = 3.1417711257934569
Average process time : 627 [ms]
n = 67108864, 52708105, 52706217, 52708105, 52706217, 52708105, 52706217, 52708105, 52706217, 52708105, 52706217, b = 527071610, pi = 3.1415916085243225
Average process time : 7300 [ms]
n = 1073741824, 843314576, 843314576, 843314576, 843314576, 843314576, 843314576, 843314576, 843314576, 843314576, 843314576, b = 8433145760, pi = 3.141591608524323
Average process time : 148791 [ms]
```

モンテカルロ法に基づくシミュレーションでは，点（x, y）を生成する試行を $n = 2^m$ 回繰り返して，四分円の内部に含まれる回数を数えることにした。そのようなシミュレーションを 10 回繰り返して，平均を取った回数から円周率 $\pi = 3.14159265359$ の近似値を求めた。また，プログラムの実行に要した時間を計測し，ミリ秒（ms）単位で表示した。

その結果を集約すると，**表 1** に示したものが得られた。シミュレーションの試行回数を増やすことに伴って，円周率 π の近似値が真の値に近づいていることが分かる。実際，1,024（$=2^{10}$）回のとき真の値との誤差が 0.01 を下回る程度であったのに対して，16,384（$=2^{14}$）回のとき真の値との誤差が 0.005 程度，

262, 144（＝2^{18}）回のとき真の値との誤差が 0. 0004 程度，4, 194, 304（＝2^{22}）回のとき真の値との誤差が 0. 0002 程度，67, 108, 864（＝2^{26}）回のとき真の値との誤差が 0. 00001 程度とシミュレーションの試行回数を増やすことによって定積分の近似値の精度が良くなっていることが読み取れる。しかし，さらに多くの試行回数でシミュレーションを繰り返しても，これ以上の精度を期待することができない。なぜならば，線形合同法による擬似乱数の周期性が影響し，

表 1　実験結果（線形合同法）

m	b	π = 3. 14159265359	error	time（ms）
10	8, 068	3. 15156250000	0. 00996984641	0
14	128, 894	3. 14682617187	0. 00523351828	2
18	2, 058, 581	3. 14114532471	−0. 00044732888	36
22	32, 943, 858	3. 14177112579	0. 00017847220	627
26	527, 071, 610	3. 14159160852	−0. 00000104507	7, 300
30	8, 433, 145, 760	3. 14159160852	−0. 00000104507	148, 791

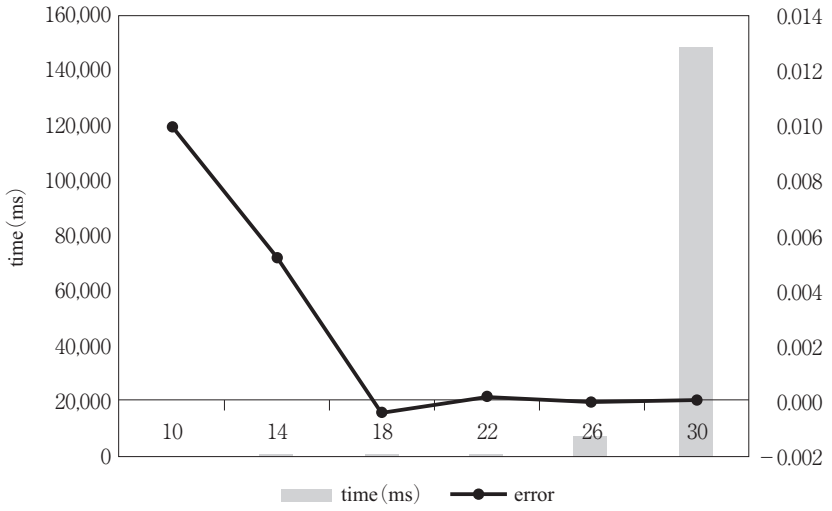

図 1　実行時間と計算精度に関するグラフ（線形合同法）

同じ点（x，y）をプロットするようになってしまうからである。

他方，シミュレーションの実行時間は，試行回数 $n = 2^m$ に比例していることが分かる。この事実は，モンテカルロ法を適用するブロックの部分で，擬似乱数利用のための関数の呼び出しと定積分計算のための条件分岐と計数処理がメインループとなることから自然な帰結である。

以上のことから，線形合同法を用いたモンテカルロ法に基づくシミュレーションにおいて，円周率 π の近似値計算は，計算コストをかけることによって途中までは精度を上げることができるものの，ある計算コストを超えると，それ以上は精度が上がらないという結果を得た。

5.2 MT 法

MT 法によって生成された擬似乱数を用いて実装したプログラムのソースコードは以下の通りである：

《ソースコード（抜粋）》

```
#include <stdio.h>
#include <time.h>
#include <stdio.h>

/* Period parameters */
#define N 624
#define M 397
#define MATRIX_A 0x9908b0dfUL     /* constant vector a */
#define UPPER_MASK 0x80000000UL   /* most significant w-r bits */
#define LOWER_MASK 0x7fffffffUL   /* least significant r bits */

static unsigned long mt[N];  /* the array for the state vector */
static int mti=N+1; /* mti==N+1 means mt[N] is not initialized */

/* initializes mt[N] with a seed */
```

```
void init_genrand(unsigned long s)
{
    mt[0]= s & 0xffffffffUL;
    for (mti=1; mti<N; mti++) {
        mt[mti] =
            (1812433253UL * (mt[mti-1] ^ (mt[mti-1] >> 30)) + mti);
        /* See Knuth TAOCP Vol2. 3rd Ed. P.106 for multiplier. */
        /* In the previous versions, MSBs of the seed affect   */
        /* only MSBs of the array mt[].                        */
        /* 2002/01/09 modified by Makoto Matsumoto             */
        mt[mti] &= 0xffffffffUL;
        /* for >32 bit machines */
    }
}

/* initialize by an array with array-length */
/* init_key is the array for initializing keys */
/* key_length is its length */
/* slight change for C++, 2004/2/26 */
void init_by_array(unsigned long init_key[], int key_length)
{
    int i, j, k;
    init_genrand(19650218UL);
    i=1; j=0;
    k = (N>key_length ? N : key_length);
    for (; k; k--) {
        mt[i] = (mt[i] ^ ((mt[i-1] ^ (mt[i-1] >> 30)) * 1664525UL))
            + init_key[j] + j; /* non linear */
        mt[i] &= 0xffffffffUL; /* for WORDSIZE > 32 machines */
        i++; j++;
        if (i>=N) { mt[0] = mt[N-1]; i=1; }
        if (j>=key_length) j=0;
    }
    for (k=N-1; k; k--){
        mt[i] = (mt[i] ^ ((mt[i-1] ^ (mt[i-1] >> 30)) * 1566083941UL))
```

```
            -i; /* non linear */
        mt[i] &= 0xffffffffUL; /* for WORDSIZE > 32 machines */
        i++;
        if (i>=N) { mt[0] = mt[N-1]; i=1; }
    }

    mt[0]=0x80000000UL; /*MSB is 1; assuring non-zero initial array*/
}

/* generates a random number on [0, 0xffffffff]-interval */
unsigned long genrand_int32(void)
{
    unsigned long y;
    static unsigned long mag01[2]={0x0UL, MATRIX_A};
    /* mag01[x] = x * MATRIX_A for x=0, 1 */

    if (mti >= N) { /* generate N words at one time */
        int kk;

        if (mti == N+1) /* if init_genrand() has not been called, */
            init_genrand(5489UL); /* a default initial seed is used*/

        for (kk=0;kk<N-M;kk++) {
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+M] ^ (y >> 1) ^ mag01[y & 0x1UL];
        }
        for(;kk<N-1;kk++) {
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+(M-N)] ^ (y >> 1) ^ mag01[y & 0x1UL];
        }
        y = (mt[N-1]&UPPER_MASK)|(mt[0]&LOWER_MASK);
        mt[N-1] = mt[M-1] ^ (y >> 1) ^ mag01[y & 0x1UL];

        mti = 0;
    }
```

```
    y = mt[mti++];

    /* Tempering */
    y ^= (y >> 11);
    y ^= (y << 7) & 0x9d2c5680UL;
    y ^= (y << 15) & 0xefc60000UL;
    y ^= (y >> 18);

    return y;
}

/* generates a random number on [0,0x7fffffff]-interval */
long genrand_int31(void)
{
    return (long)(genrand_int32()>>1);
}

/* generates a random number on [0,1]-real-interval */
double genrand_real1(void)
{
    return genrand_int32()*(1.0/4294967295.0);
    /* divided by 2^32-1 */
}

/* generates a random number on [0,1)-real-interval */
double genrand_real2(void)
{
    return genrand_int32()*(1.0/4294967296.0);
    /* divided by 2^32 */
}

/* generates a random number on (0,1)-real-interval */
double genrand_real3(void)
{
    return (((double)genrand_int32()) + 0.5)*(1.0/4294967296.0);
```

```
    /* divided by 2^32 */
}

/* generates a random number on [0,1) with 53-bit resolution*/
double genrand_res53(void)
{
    unsigned long a=genrand_int32()>>5, b=genrand_int32()>>6;
    return(a*67108864.0+b)*(1.0/9007199254740992.0);
}
/* These real versions are due to Isaku Wada, 2002/01/09 added */

int main( int argc, char *argv[] )
{
    long i, j;
    double x, y;
    long b, c, n = 1024;
    clock_t start, end;

    /* Initialization */
    printf("*** Monte Carlo pi Computation ***¥n");

    /* Monte Carlo Simulation */
    while( n > 0 ){
        b = 0;
        printf( "n = %d, ", n );
        start = clock();
        for( j = 0; j < 10; j++ ){
            c = 0;
            for( i = 0; i < n; i++ ){
                x = genrand_real2();
                y = genrand_real2();
                if( x * x + y * y < 1 ){
                    c++;
                }
            }
```

```
            b += c;
            printf( "%d, ", c );
        }
        end = clock();
        printf( "b = %d, pi = %18.16f¥n", b, 4.0 * b / n / 10.0 );
        printf( "Average process time : %d [ms]¥n", ( end - start ) / 10 );
        n *= 16;
    }

    return 0;
}
```

上記ソースコードの実行結果は，以下の通りである：

《実験結果》

```
C:¥Users¥dan>pi_mt
*** Monte Carlo pi Computation ***
n = 1024, 820, 805, 806, 803, 809, 795, 806, 810, 823, 806, b = 8083, pi =
3.1574218749999998
Average process time : 0 [ms]
n = 16384, 12865, 12910, 12938, 12883, 12967, 12758, 12875, 12818, 12860, 12871, b
= 128745, pi = 3.1431884765625000
Average process time : 1 [ms]
n = 262144, 205780, 205724, 205867, 205659, 205864, 205862, 205740, 205576,
206020, 205873, b = 2057965, pi = 3.1402053833007813
Average process time : 23 [ms]
n = 4194304, 3294444, 3295877, 3296810, 3294180, 3294596, 3292699, 3292549,
3293584, 3294946, 3295358, b = 32945043, pi = 3.1418841361999510
Average process time : 393 [ms]
n = 67108864, 52710691, 52712781, 52709092, 52707497, 52703361, 52706627,
52707235, 52711958, 52705211, 52705218, b = 527079671, pi = 3.1416396558284760
Average process time : 6275 [ms]
```

```
n = 1073741824, 843328880, 843308843, 843289060, 843336991, 843310709, 843349300,
843327262, 843341006, 843331069, 843314140, b = -156697332, pi = -0.0583743050694466
Average process time : 95540 [ms]
```

その結果を集約すると，**表 2** に示したものが得られた。シミュレーションの試行回数を増やすことに伴って，線形合同法を用いたときと同様に，円周率 π

表 2　実験結果（MT 法）

m	b	$\pi = 3.14159265359$	error	time(ms)
10	8, 083	3. 15742187500	0. 01582922141	0
14	128, 745	3. 14318847656	0. 00159582297	1
18	2, 057, 965	3. 14020538330	−0. 00138727029	23
22	32, 945, 043	3. 14188413620	0. 00029148261	393
26	527, 079, 671	3. 14163965583	0. 00004700224	6, 275
30	8, 433, 237, 260	3. 14162569493	0. 00003304134	95, 540

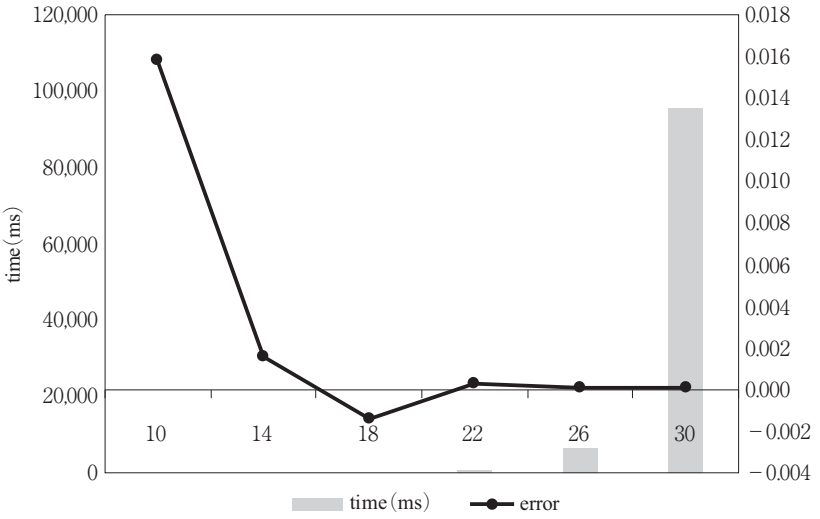

図 2　実行時間と計算精度に関するグラフ（MT 法）

の近似値が真の値に近づいていることが分かる。実際，1, 024（＝2^{10}）回のとき真の値との誤差が 0. 016 程度であったのに対して，16, 384（＝2^{14}）回のとき真の値との誤差が 0. 002 程度，262, 144（＝2^{18}）回のとき真の値との誤差が −0. 002 程度，4, 194, 304（＝2^{22}）回のとき真の値との誤差が 0. 0003 程度，67, 108, 864（＝2^{26}）回のとき真の値との誤差が 0. 00005 程度，1, 073, 741, 824（＝2^{30}）回のとき真の値との誤差が 0. 00003 程度とシミュレーションの試行回数を増やすことによって定積分の近似値の精度が良くなっていることが読み取れる。

5. 3　量子乱数

今回の研究テーマである量子乱数によって生成された真正乱数を用いて実装したプログラムのソースコードは以下の通りである：

《ソースコード（抜粋)》

```
#include <float.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#if defined(__unix__) || defined(__APPLE__)
#include <unistd.h>
#else
#include "getopt.h"
#endif

#include "Quantis.h"

#ifdef _WIN32
# pragma comment(lib, "Quantis.lib")
```

```
#endif

#define BYTES_DEFAULT   10
#define BYTES_MAX       100000

static void printUsage()
{
   printf
      ("Usage : qrng [-i|-p cardNumber|-u cardNumber] [-n bytes]¥n¥n");
   printf("Options¥n");
   printf("-h : display help screen¥n");
   printf("-i : display information on all cards found¥n");
   printf("-n : set the number of bytes to read(default %d, max %d)¥n",
      BYTES_DEFAULT, BYTES_MAX);
   printf("-p : set the PCI card number¥n");
   printf("-u : set the USB card number¥n");
}

static void _printCardsInfo(QuantisDeviceType deviceType)
{
   float driverVersion = 0;
   int devicesCount = 0;
   int i = 0;
   int j = 0;

   /* Driver version */
   driverVersion = QuantisGetDriverVersion(deviceType);
   if (driverVersion < 0.0f)
   {
      fprintf(stderr,
         "Error while getting driver version: %s¥n",
         QuantisStrError((int)driverVersion));
   return ;
   }
   printf("Using driver version %f¥n",
```

```
    driverVersion);

/* Devices count */
devicesCount = QuantisCount(deviceType);
printf("Found %d card(s)¥n", devicesCount);

/* Devices details */
for (i = 0; i < devicesCount; i++ )
{
    int boardVersion;
    int power;

    /* Display device's info */
    printf("-Details for device #%d:¥n", i);

    /* Board version */
    boardVersion = QuantisGetBoardVersion(deviceType, i);
    if (boardVersion < 0)
    {
        printf( "core version: error (%s)¥n",
            QuantisStrError(boardVersion));
    }
    else
    {
        printf("core version: 0x%08X¥n", boardVersion);
    }

    /* Serial number */
    printf( "serial number: %s¥n",
        QuantisGetSerialNumber(deviceType, i));

    /* Manufacturer */
    printf( "manufacturer: %s¥n",
        QuantisGetManufacturer(deviceType, i));
```

```
/* Modules power */
printf("    module(s) powered: ");
power = QuantisGetModulesPower(deviceType, i);
if (power == 0)
{
    printf("no¥n");
}
else if (power > 0)
{
    printf("yes¥n");
}
else
{
    printf("error (%s)¥n", QuantisStrError(power));
}

/* Display device's modules info */
for (j = 0; j < 4; j++)
{
    int result;
    char* strMask = NULL;
    char* strStatus = NULL;

    result = QuantisGetModulesMask(deviceType, i);
    if (result < 0)
    {
        strMask = "error while retrieving mask";
        strStatus = "";
    }
    else if (result & (1 << j))
    {
        strMask = "found";
        result = QuantisGetModulesStatus(deviceType, i);
        if (result < 0)
        {
```

```
                strStatus = "(error while retrieving status)";
            }
            else if (result & (1 << j))
            {
                strStatus = "(enabled)";
            }
            else
            {
                strStatus = "(disabled)";
            }
        }
        else
        {
            strMask = "not found";
            strStatus = "";
        }
        printf("module %d: %s %s¥n", j, strMask, strStatus);
    }
  }
}

static void printCardsInfo()
{
  printf("Displaying cards info:¥n");

  printf("¥n* Searching for PCI devices...¥n");
  _printCardsInfo(QUANTIS_DEVICE_PCI);

  printf("¥n* Searching for USB devices...¥n");
  _printCardsInfo(QUANTIS_DEVICE_USB);
}


static void printRandomData(QuantisDeviceType deviceType,
  int cardNumber, int bytesNum)
```

```
{
    int i;
    double x, y;
    int b = 0, c = 0, n = 10000;

    for( i = 0; i < n; i++ ){
        QuantisReadDouble_01( deviceType, cardNumber, &x );
        QuantisReadDouble_01( deviceType, cardNumber, &y );
        if( x * x + y * y < 1 ){
            c++;
        }
        else if( x * x + y * y == 1 ){
            b++;
        }
    }

    printf( "b = %d, c = %d¥n", b, c );
    printf( "pi = %f¥n", 4.0 * c / n );
    printf( "pi = %f¥n", 4.0 * ( c + b ) / n );

    return;
}


int main( int argc, char *argv[] )
{
    QuantisDeviceType deviceType = QUANTIS_DEVICE_PCI;
    int cardNumber = 0;

    long i, j;
    double x, y;
    unsigned int xx, yy;
    long b, c, n = 1024;
    clock_t start, end;
    const unsigned int m = 2 << 31 - 1;
```

```
/* Initialization */
printf("*** Monte Carlo pi Computation ***¥n");

deviceType = QUANTIS_DEVICE_USB;
cardNumber = 0;

/* Monte Carlo Simulation */
while( n > 0 ){
    if( cardNumber >= 0 ){
        b = 0;
        printf( "n = %d, ", n );
        start = clock();
        for( j = 0; j < 10; j++ ){
            c = 0;
            for( i = 0; i < n; i++ ){
                QuantisReadInt( deviceType, cardNumber, &xx );
                QuantisReadInt( deviceType, cardNumber, &yy );
                x = (double)( xx % m ) / m;
                y = (double)( yy % m ) / m;
                if( x * x + y * y < 1 ){
                    c++;
                }
            }
            b += c;
            printf( "%d, ", c );
        }

        end = clock();
        printf( "b = %d, pi = %18.16f¥n", b, 4.0 * b / n / 10.0 );
        printf("Average process time : %d [ms]¥n", (end - start) / 10);
    }

    n *= 16;
}
```

```
    return QUANTIS_SUCCESS;
}
```

上記ソースコードの実行結果は，以下の通りである：

《実験結果》

```
C:¥Users¥dan>pi_q
*** Monte Carlo pi Computation ***
n = 1024, 807, 805, 811, 811, 797, 804, 816, 792, 825, 808, b = 8076, pi =
3.1546875000000001
Average process time : 5065 [ms]
n = 16384, 12831, 12878, 12842, 12984, 12918, 12887, 12865, 12851, 12856, 12784, b
= 128696, pi = 3.1419921875000001
Average process time : 78964 [ms]
n = 262144, 205735, 205881, 205701, 205777, 205467, 205771, 205799, 205555,
206026, 206289, b = 2058001, pi = 3.1402603149414063
Average process time : 1205193 [ms]
n = 4194304, 3294872, 3293895, 3293669, 3294316, 3294805, 3294509, 3293605,
3294035, 3293594, 3293052, b = 32940352, pi = 3.1414367675781252
Average process time : 20833224 [ms]
n = 67108864, 52708872, 52710537, 52707634,
```

その結果を集約すると，**表 3** に示したものが得られた。シミュレーションの試行回数を増やすことに伴って，これまでの擬似乱数を用いたときと同様に，円周率 π の近似値が真の値に近づいていることが分かる。実際，1,024（$=2^{10}$）回のとき真の値との誤差が 0.01 程度であったのに対して，16,384（$=2^{14}$）回のとき真の値との誤差が 0.0004 程度，262,144（$=2^{18}$）回のとき真の値との誤差が -0.001 程度，4,194,304（$=2^{22}$）回のときは，真の値との誤差が -0.0002 程度とシミュレーションの試行回数を増やすことによって，多少の増減はあるものの，定積分の近似値の精度が良くなっていることが読み取れる。

しかし，このプログラムの実行には，14 日以上の計算時間がかかってしまい，途中で計算を打ち切るという残念な結果に終わってしまった。量子デバイスによって生成される乱数は，その量子力学に基づく観測と生成という動作原理であることから，それほど生成速度に期待することはできないと考えることが自然である。

表 3　実験結果（量子乱数）

m	b	$\pi = 3.14159265359$	error	time（ms）
10	8, 076	3. 15468750000	0. 01309484641	5, 065
14	128, 696	3. 14199218750	0. 00039953391	78, 964
18	2, 058, 001	3. 14026031494	−0. 00133233865	1, 205, 193
22	32, 940, 352	3. 14143676758	−0. 00015588601	20, 833, 224
26	158, 127, 043	＊3. 14170207580	0. 00010942221	−

＊計算時間が 14 日を超えたため，それまでの 3 試行分について平均を求めた。

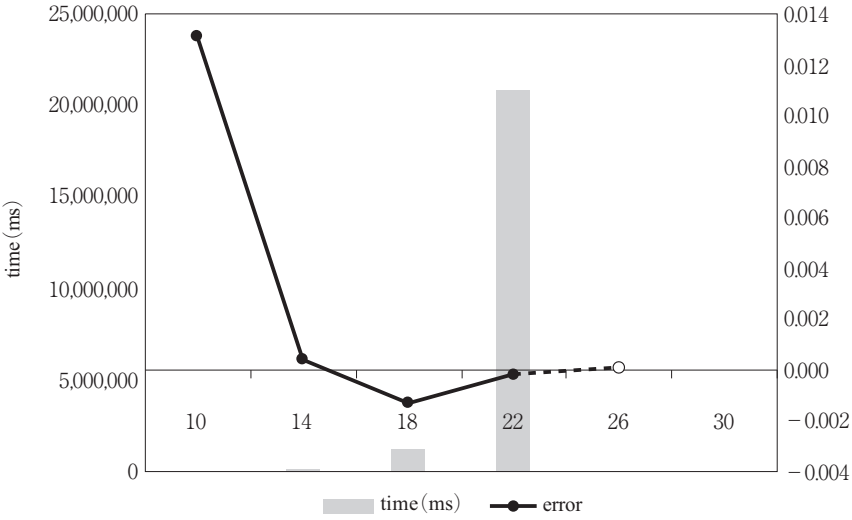

図 3　実行時間と計算精度に関するグラフ（量子乱数）

6 ま と め

本稿では，量子デバイスとして実用化された量子乱数発生器 IDQ-Quantis について，ランダムビットストリームの生成速度とともに実際のアプリケーションに取り入れたときの計算性能を評価した。具体的には，すでに解析的な計算結果が得られている定積分を取り上げ，その公式から円周率が計算できることに着目し，ランダムビットストリームを用いて円周率 π の近似値を計算するというベンチマークを行った。

その結果，標準で毎秒 4 M ビット（最大で 16 M ビット）のランダムビットストリームの生成速度は，線形合同法やメルセンヌ・ツイスター法に基づく擬似乱数に比べて遅いということが明らかになった。もちろん，物理乱数という性質上，量子乱数の周期は理論的に無限大であって，シミュレーションの試行回数の増大に対して有効であるということに変わりはない。

モンテカルロ法を適用するモデルに依存することではあるが，擬似乱数の周期性が現れたとしても，その網羅性によって近似値を求めることを指向するシミュレーションの実行時には，線形合同法による擬似乱数生成が最も有力である。また，周期性が問題になるようなモデルに対しては，メルセンヌ・ツイスター法による擬似乱数生成を用いることによって，計算速度という点では劣る面があるにせよ，計算コストをかけることによって，線形合同法を用いる場合に比べて，精度を向上させることが可能になる。

量子乱数は，その量子力学に基づく観測と生成という動作原理であることから，生成速度に期待することはできない。しかし，無限大の周期という真正乱数が生成されるという点は大きな特徴であって，シミュレーションにリアルタイム性が求められないような場合であれば，計算時間をかければかけるだけ精度の向上が見込めることから，スーパーコンピュータを併用した計算などハイパフォーマンスコンピューティング（HPC）の分野で応用される余地が考えられる。その反面，現在の一般的なコンピュータの計算能力では，量子乱数の性

能を最大限に利用できないことを意味している。

モンテカルロ法に基づくシミュレーションなど大規模で高速な演算能力が求められるアプリケーションでは実用的ではないものの，量子乱数は情報セキュリティ分野における暗号用に用いることは，擬似乱数には実現できない信頼性を発揮する。線形合同法やメルセンヌ・ツイスター法など一般の線形アルゴリズムに基づく擬似乱数では，暗号通信に用いる乱数として安全なものが生成できるわけではない。計算量理論によると，漸化式を用いる乱数生成法では，十分な長さの出力列を解析することによって乱数の出方が見破られるという問題が発生してしまう。そこで，どのような解析にも破られないという性質のある量子乱数は，情報セキュリティの分野で最大の能力を発揮することが理解される。量子乱数の応用事例としては，暗号化を含む情報セキュリティが最も有望であることが明らかとなった。

なお，本研究は 2014 年度に交付を受けた松山大学特別研究助成の成果の一部である。

参 考 文 献

［1］Y. Dan,“Mathematical Analysis and Simulation of Information Diffusion on Networks,” SAINT 2011 Workshop : IT enabled Services, IEEE Computer Society, pp. 550-555.（2011）

［2］Y. Dan,“Modeling and Simulation of Diffusion Phenomena on Social Networks,” IEEE Proceedings of ICCMS 2011, Vol. 1, pp. 139-146.（2011）

［3］Y. Dan and T. Moriya,“Analysis and Simulation of Power Law Distribution of File Types in File Sharing Systems,” The Proceedings of SIMUL 2011, IARIA, pp. 116-121.（2011）

［4］檀裕也「モンテカルロ法による研究室配属モデルのシミュレーション」松山大学論集，第 19 巻第 4 号，pp. 75-90.（2007 年 11 月）

［5］檀裕也「モンテカルロ・シミュレーションによる予測の精緻化に関する数理モデル」松山大学論集，第 23 巻第 3 号，pp. 315-334.（2011 年 8 月）

［6］J. von Neumann, Various Techniques used in Connection with Random Digit Monte Carlo, Nat. Bureau Standards, 12（1951）, pp. 36-38.

［7］Nicholas Metropolis and Stanislaw Ulam, The Monte Carlo Method, Journal of the American Statistical Association, 44, 335-341.（1949）

［8］株式会社アルゴ IDQ-QUANTIS,
http://www.argocorp. com/compo/IDQ/IDQ.html
［9］Mersenne Twister Home Page,
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/mt.html
［10］M. Matsumoto and T. Nishimura, "Mersenne Twister : A 623-dimensionally equidistributed uniform pseudorandom number generator", ACM Trans. on Modeling and Computer Simulation Vol. 8, No. 1, January pp. 3-30（1998）DOI : 10. 1145/272991. 272995
（以上，URL は 2015 年 3 月 10 日閲覧）